HMVFS: A Hybrid Memory Versioning File System

Shengan Zheng, Linpeng Huang, Hao Liu, Linzhu Wu, Jin Zha

Department of Computer Science and Engineering

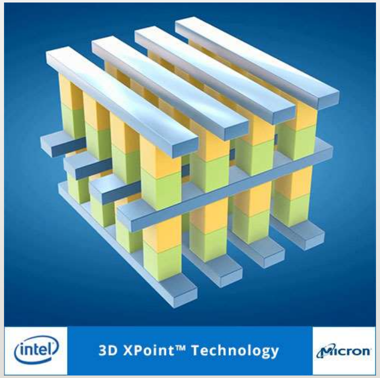
Shanghai Jiao Tong University

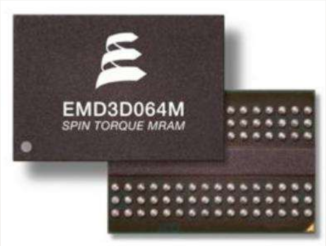
Outline

- Introduction
- Design
- Implementation
- Evaluation
- Conclusion

Introduction

- Emerging Non-Volatile Memory (NVM)
  - Persistency as disk
  - Byte addressability as DRAM
- Current file systems for NVM
  - PMFS, SCMFS, BPFS
  - Non-versioning, unable to recover old data
- Hardware and software errors
  - Large dataset and long execution time
  - Fault tolerance mechanism is needed
- Current versioning file systems
  - BTRFS, NILFS2
  - Not optimized for NVM





Design Goals

- Strong consistency
  - A Stratified File System Tree (SFST) represents the snapshot of whole file system
  - Atomic snapshotting is ensured
- Fast recovery
  - Almost no redo or undo overhead in recovery
- High performance
  - Utilize the byte-addressability of NVM to update the tree metadata at the granularity of bytes
  - Log-structured updates to files balance the endurance of NVM
  - Avoid write amplification
- User friendly
  - Snapshots are created automatically and transparently

Overview

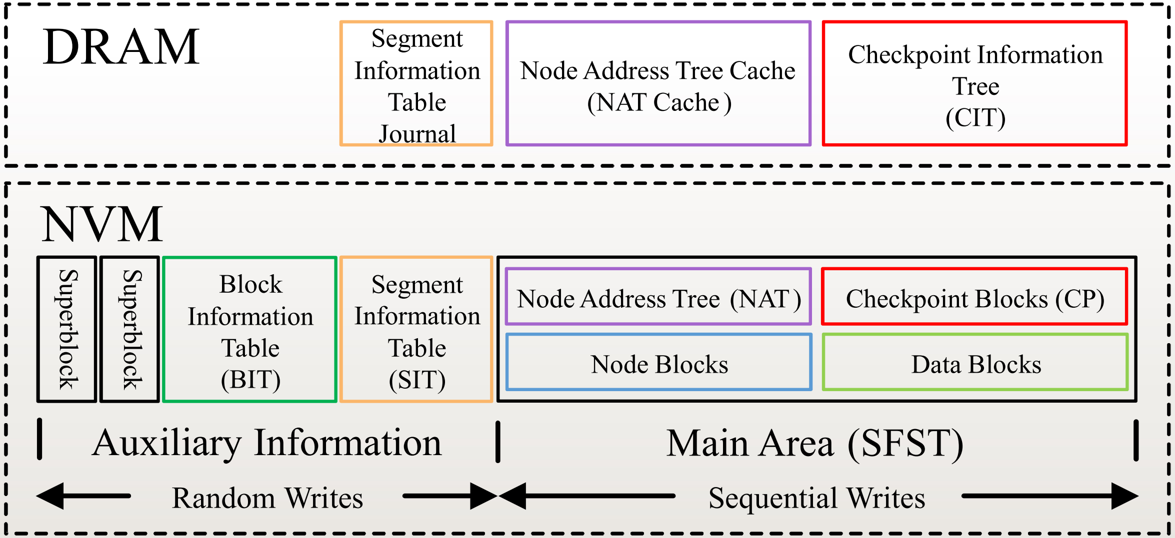
- HMVFS is an NVM-friendly log-structured versioning file system
- Space-efficient file system snapshotting
- HMVFS decouples tree metadata from tree data
- High performance and consistency guarantee
- POSIX compliant

Outline

- Introduction
- Design
- Implementation
- Evaluation
- Conclusion

On-Memory Layout

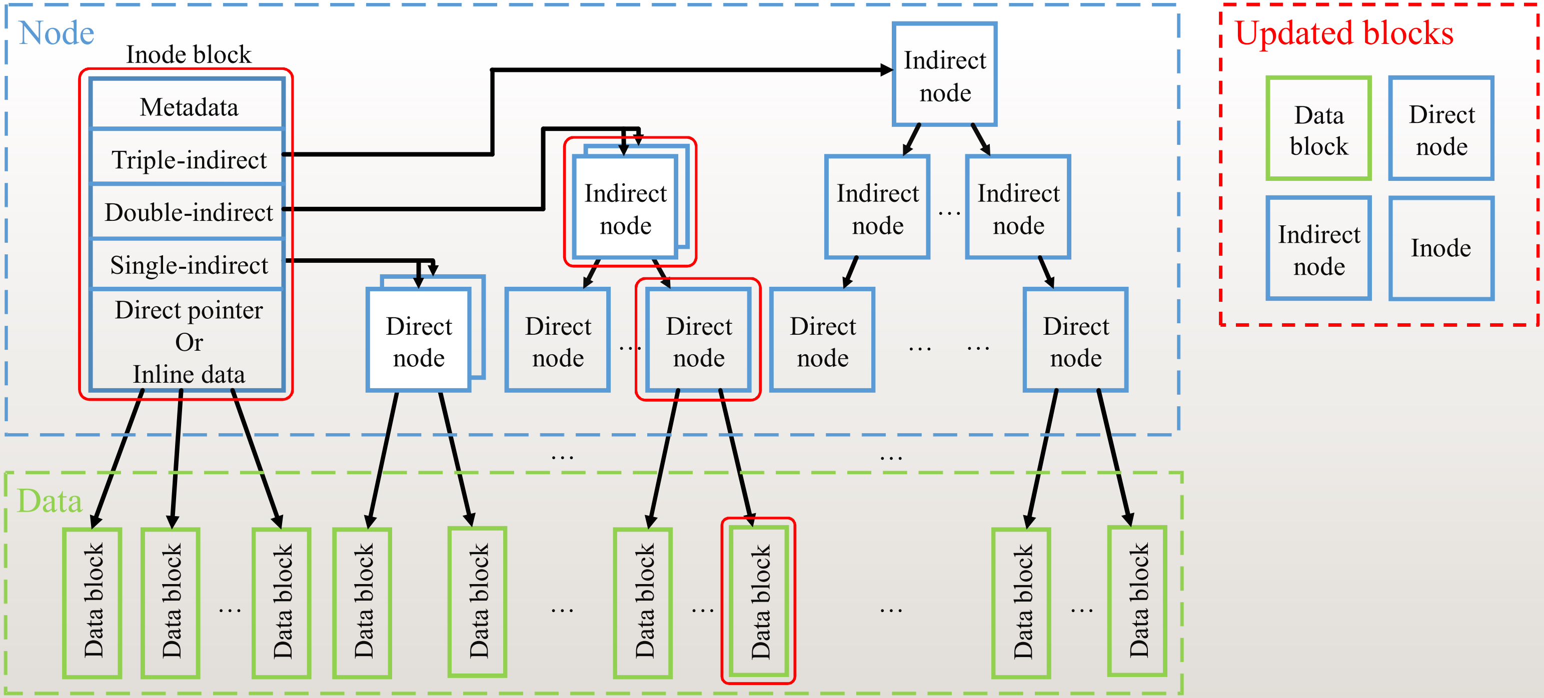
- DRAM: cache and journal



- NVM:
  - Random write zone
    - File system metadata
    - Tree metadata
  - Sequential write zone
    - File metadata and data
    - Tree data

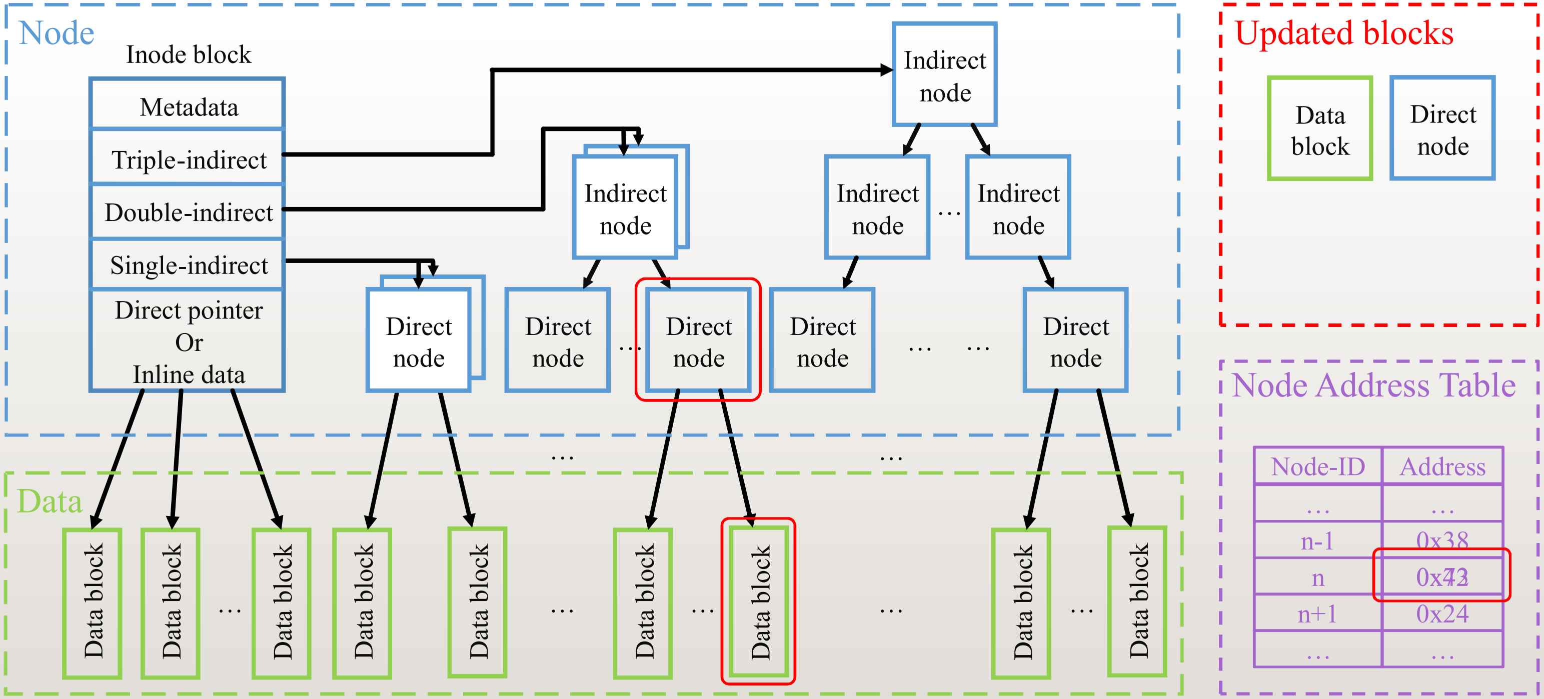
Index Structure in traditional Log-structured File Systems

- Update propagation problem



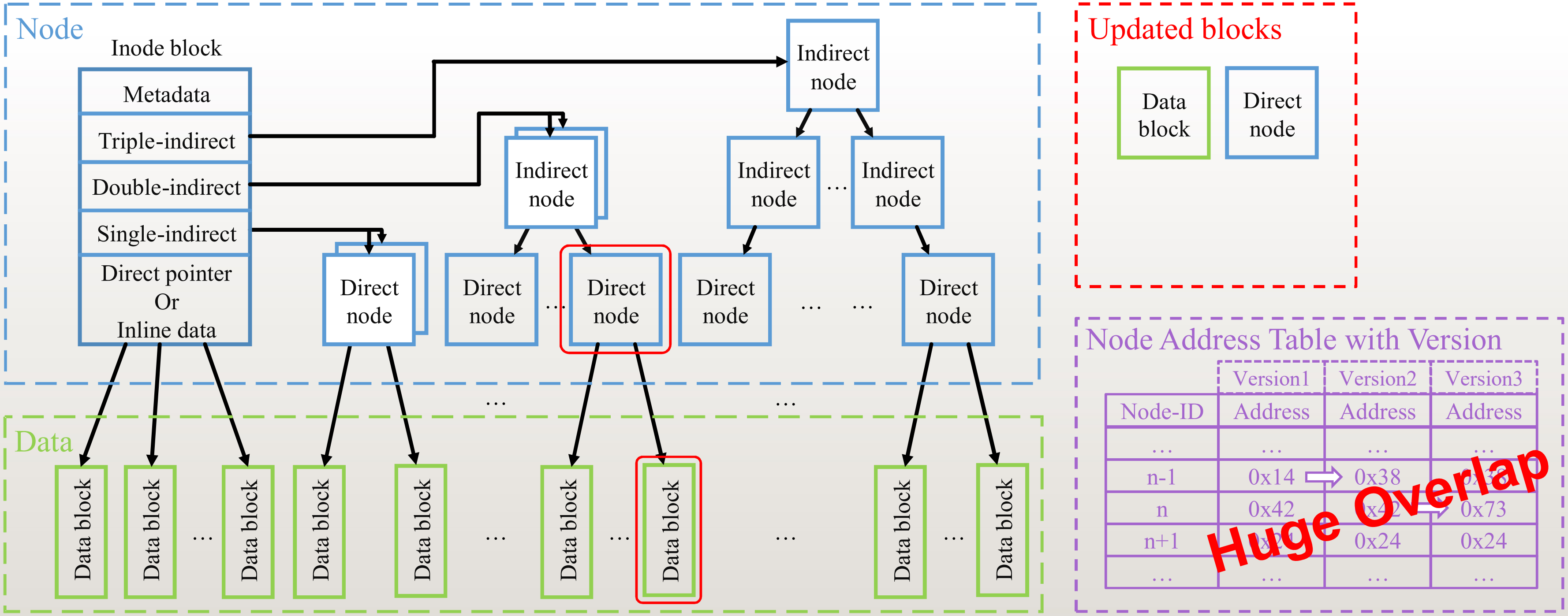
Index Structure without write amplification

- Node Address Table



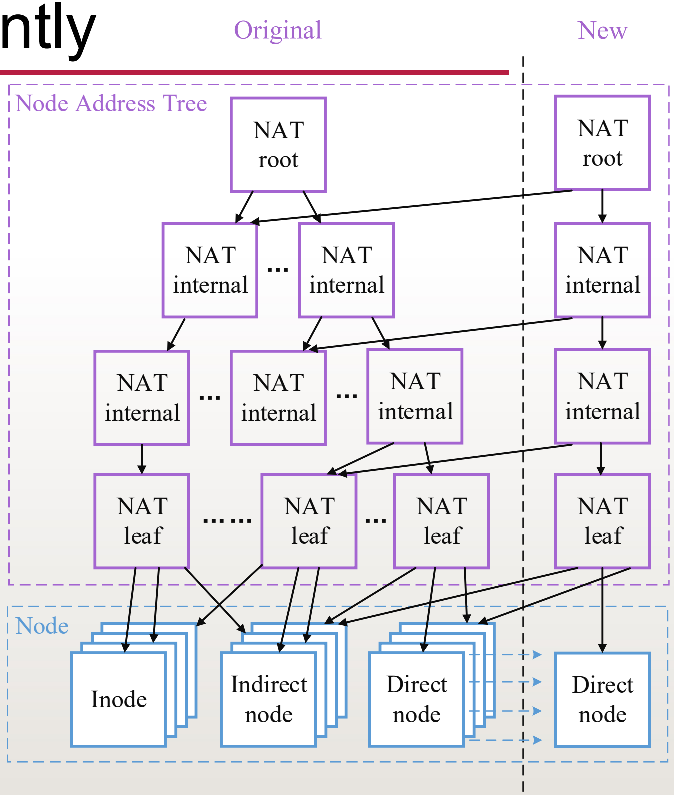
Index Structure for versioning

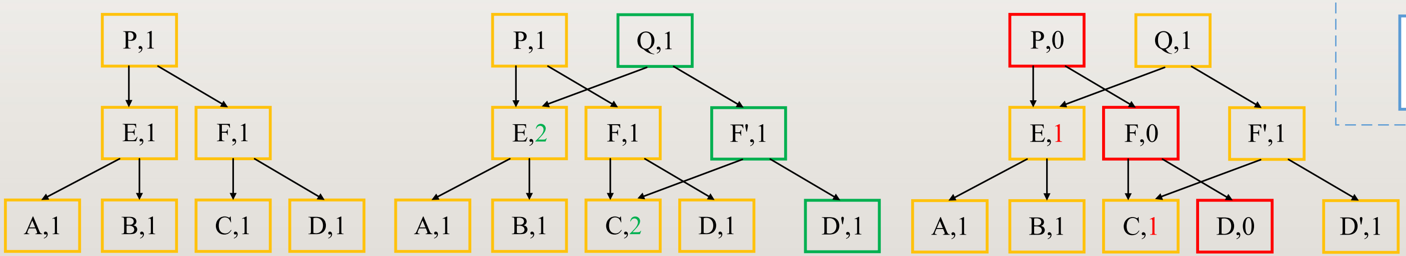
- Node Address Table with the dimension of version.



How to store different trees space-efficiently

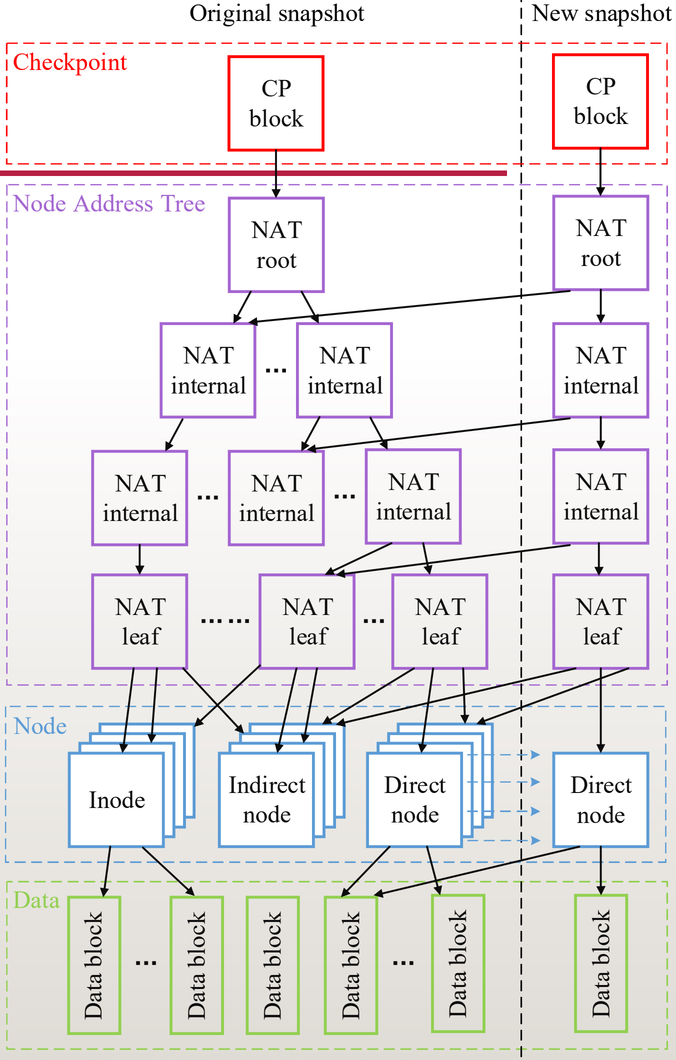
- Node Address Tree (NAT)
  - A four-level B-tree to store multi-version Node Address Table space-efficiently
  - Adopt the idea of CoW friendly B-tree
  - NAT leaves contain NodeID-address pairs
  - Other tree blocks in NAT contain pointers to lower level blocks.





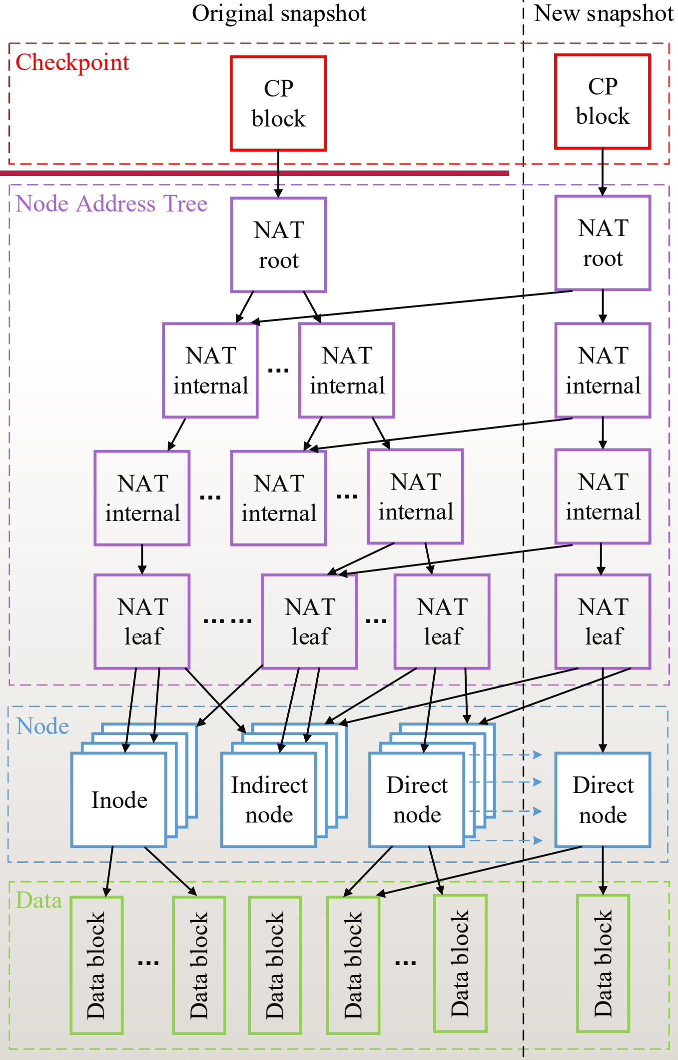
Stratified File System Tree (SFST)

- Four different categories of blocks:
  - Checkpoint layer
  - Node Address Tree (NAT) layer
  - Node layer
  - Data layer
- All blocks from SFST are stored in the main area with log-structured writes
  - Balance the endurance of NVM media
- Each SFST represents a valid snapshot of file system
  - Share overlapped blocks to achieve space-efficiency



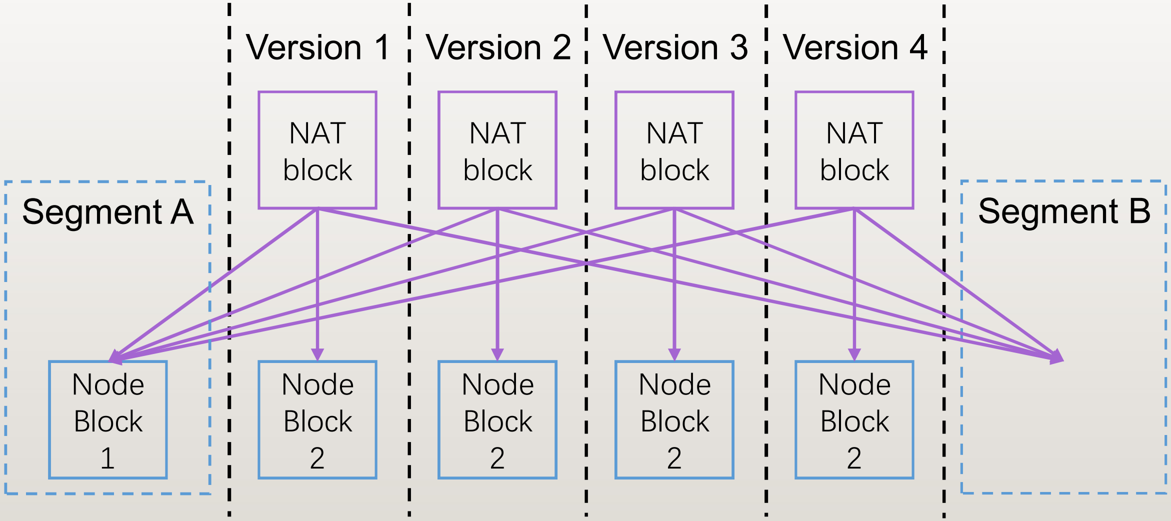
Stratified File System Tree (SFST)

- The metadata of SFST
  - In auxiliary information zone
  - Random write updates
- Segment Information Table (SIT)
  - Contains the status information of every segment
- Block Information Table (BIT)
  - Keeps the information of every block
  - Update precisely at variable bytes granularity
  - Contains:
    - Start and end version number
    - Block type
    - Node ID
    - Reference count



Garbage Collection in HMVFS

- Move all the valid blocks in the victim segment to the current segment
- When finished, update SIT and create a snapshot
- Handle block sharing problem



Outline

- Introduction
- Design
- Implementation
- Evaluation
- Conclusion

Block Information Table (BIT)

- Block sharing problem
  - The corresponding pointer in the parent block must be updated if a new child block is written in the main area
- Node ID and block type
  - Used to locate parent node

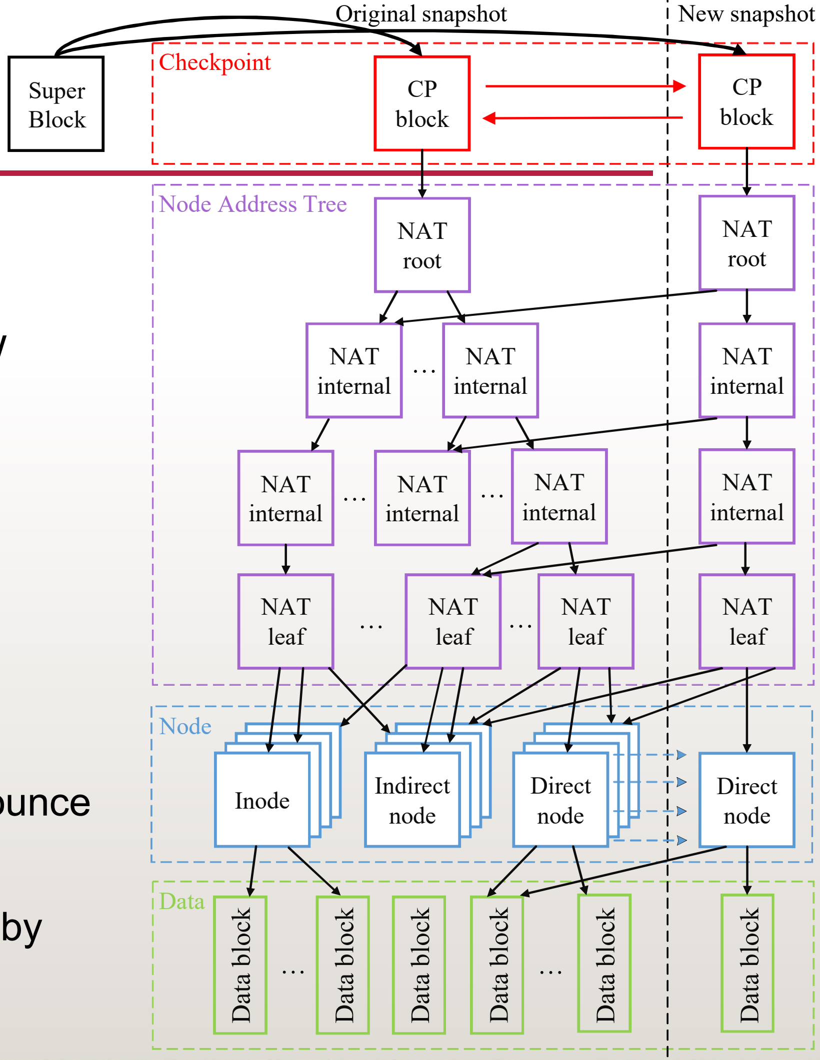
Type of the block	Type of the parent	Node ID
Checkpoint	N/A	N/A
NAT internal	NAT internal	Index code in NAT
NAT leaf		
Inode	NAT leaf	Node ID
Indirect		
Direct		
Data	Inode or direct	Node ID of parent node

Block Information Table (BIT)

- Start and end version number
  - The first and last versions in which the block is valid
  - Operations like write and delete set these two variables to the current version number
- Reference count
  - The number of parent nodes which are linked to the block
  - Update with lazy reference counting
  - File level operations and snapshot level operations update the reference count
  - If the count reaches zero, the block will become garbage

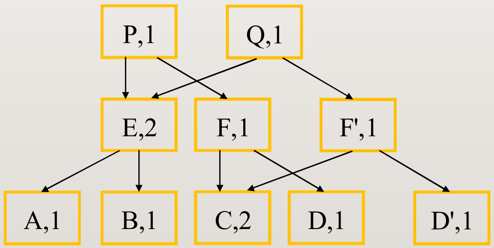
Snapshot Creation

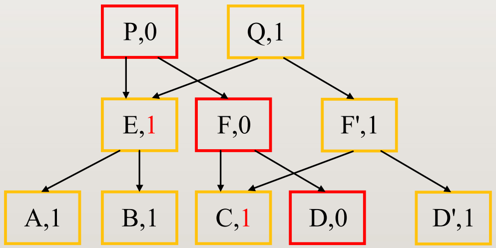
- Strong consistency is guaranteed
- Flush dirty NAT entries from DRAM to form a new Node Address Tree
  - Follow the bottom-up procedure
- Status information are stored in checkpoint block
- Space-efficient snapshot
- The atomicity of snapshot creation is ensured
  - Atomic update to the pointer in superblock to announce the validity of the new snapshot
  - Crash during snapshot creation can be recovered by undo or redo depend on the validity



Snapshot Deletion

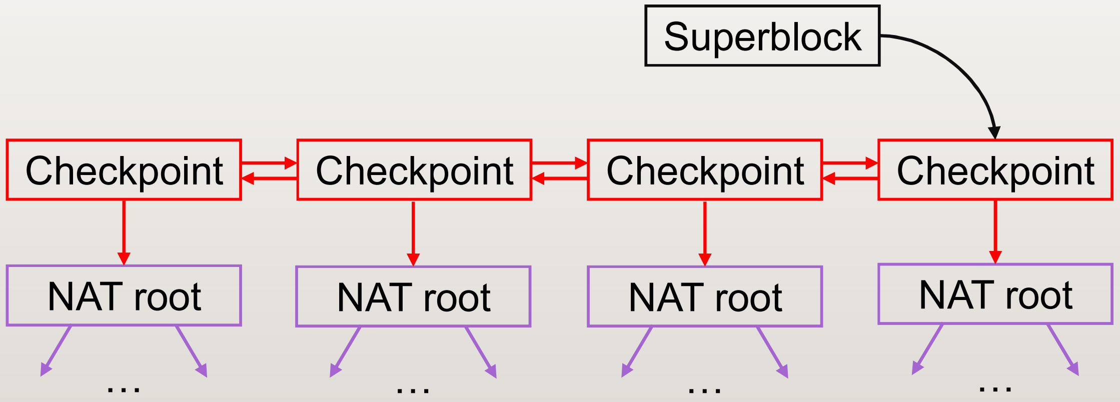
- Deletion starts from the checkpoint block
  - Checkpoint cache is stored in DRAM
  - Follows the top-down procedure to decrease reference counts
  - Consistency is ensured by journaling
- Call garbage collection afterwards
  - Many reference counts have decreased to zero





Crash Recovery

- Mount the writable last completed snapshot
  - No additional recovery overhead
- Mount the read-only old snapshots
  - Locate the checkpoint block of the snapshot
  - Retrieve files via SFST

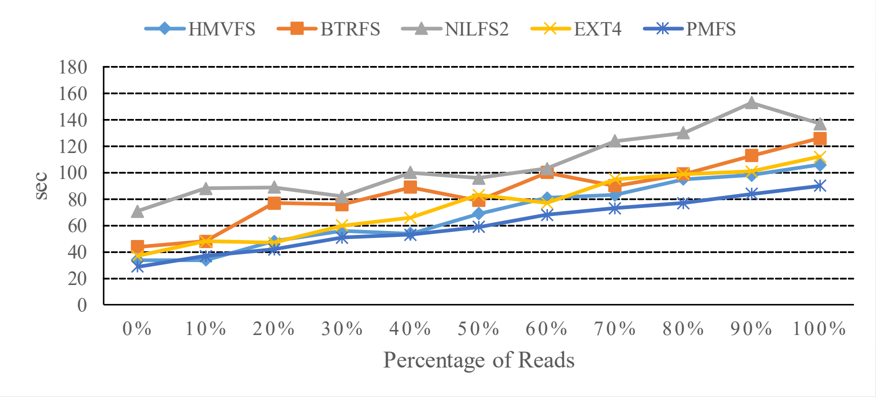


Outline

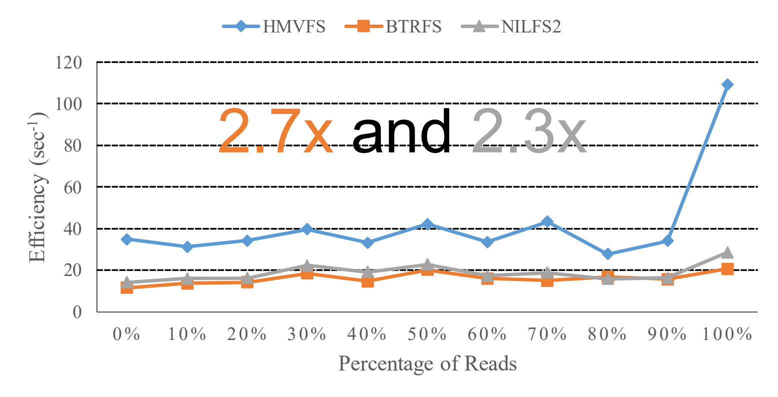
- Introduction
- Design
- Implementation
- Evaluation
- Conclusion

Evaluation

- Experimental Setup
  - A commodity server with 64 Intel Xeon 2GHz processors and 512GB DRAM
  - Performance comparison with PMFS, EXT4, BTRFS, NILFS2
- Postmark results
  - Different read bias numbers



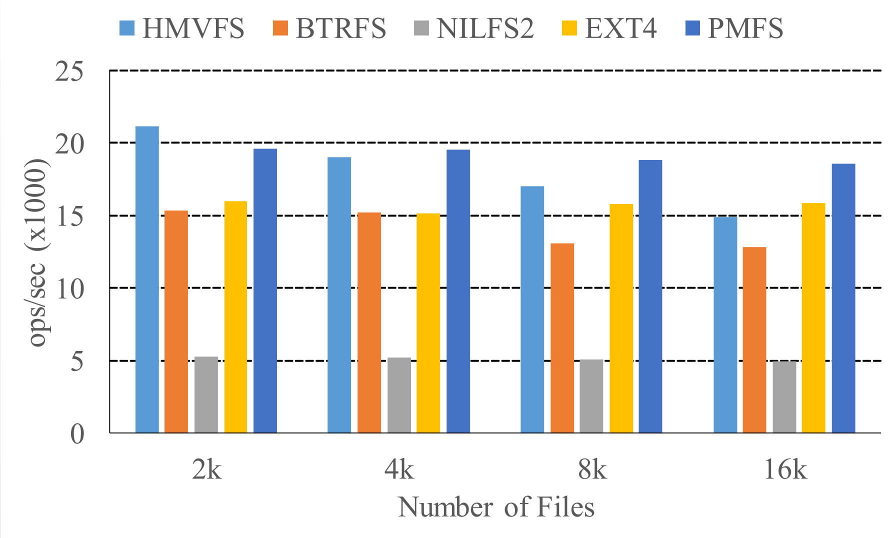
Transaction performance



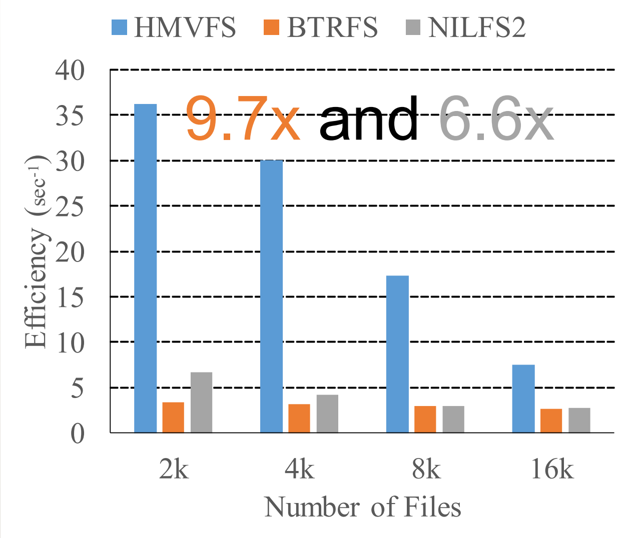
Snapshotting efficiency

Evaluation

- Filebench results
  - Fileserver
  - Different numbers of files



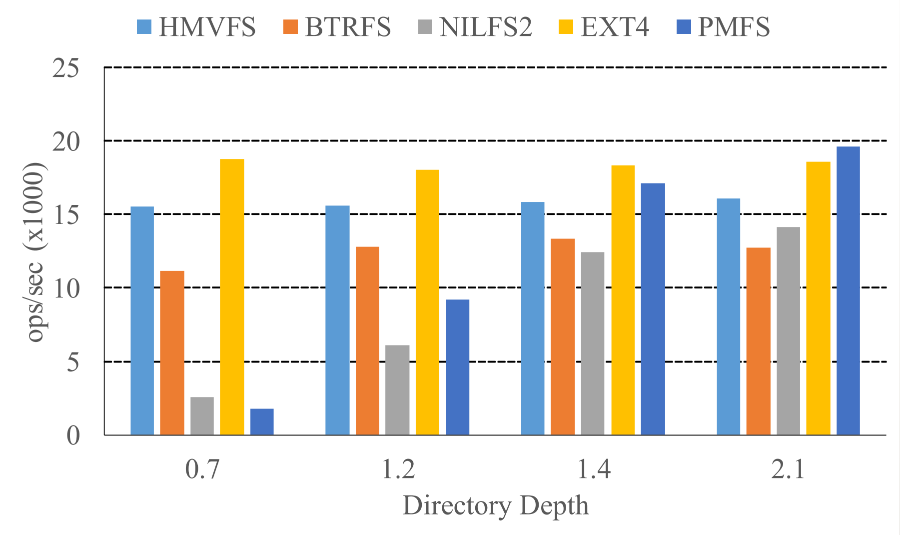
Throughput performance



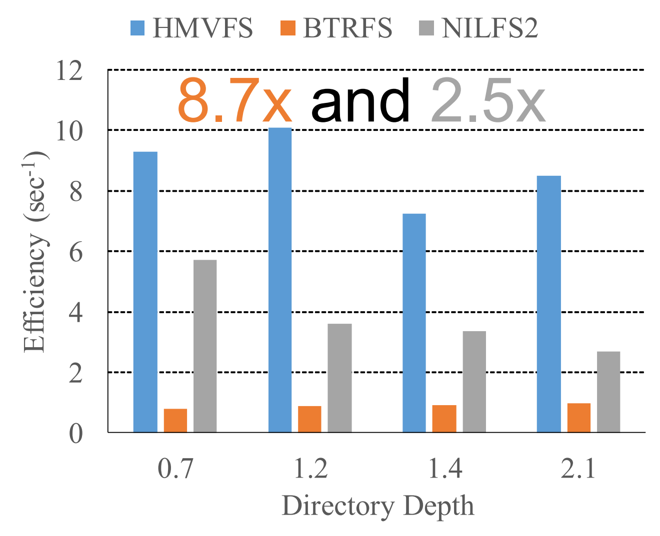
Snapshotting efficiency

Evaluation

- Filebench results
  - Varmail
  - Different depths of directories



Throughput performance



Snapshotting efficiency

Outline

- Introduction
- Design
- Implementation
- Evaluation
- Conclusion

Conclusion

- HMVFS is the first file system to solve the consistency problem for NVM-based in-memory file systems using snapshotting.
- Metadata of the Stratified File System Tree (SFST) is decoupled from data and is updated at byte granularity
- HMVFS stores the snapshots space-efficiently with shared blocks in SFST and handles write amplification problem and block sharing problem well
- HMVFS exploits the structural benefit of CoW friendly B-tree and the byte-addressability of NVM to automatically take frequent snapshots
- HMVFS outperforms tradition versioning file systems in snapshotting and performance while providing strong consistency guarantee and having little impact on foreground operations

- Q & A
- Thank you